Language with Variable Local Binding

Environment Version · AST · Value · Env · Parser · Interpreter · Runner

---

1. Grammar (BNF)

The surface language supports numeric literals, addition, a single-variable local binding form, and variable references.

```
&lt;expr&gt; ::= &lt;num&gt;
       | {+ &lt;expr&gt; &lt;expr&gt;}
       | {let1 {&lt;var&gt; &lt;expr&gt;} &lt;expr&gt;}
       | &lt;var&gt;
```

2. AST

The **Exp** type represents all possible expressions. Constructors are suffixed with **E** to distinguish expressions from values.

```
(define-type Exp
  [numE  (n    : Number)]
  [plusE (left : Exp) (right : Exp)]
  [varE  (name : Symbol)]
  [let1E (var  : Symbol)
         (val  : Exp)
         (body : Exp)])
```

3. Value Type

The **Value** type represents what the interpreter returns. Unlike the substitution version, the environment interpreter always returns a **Value**, not a raw number.

```
(define-type Value
  [numV (n : Number)])
```

4. Environment

An **Env** is a hash table mapping Symbols to Values. It is treated as *immutable*: **extend** always produces a new Env without mutating the original.

```
(define-type-alias Env (Hashof Symbol Value))

(define mt-env : Env (hash empty))   ; the empty environment
```

5. lookup (Symbol Env -> Value)

Searches the environment for a variable name. Returns its value if found; raises an error if the variable is unbound. **hash-ref** returns an **(Optionof Value)**, so both cases must be handled explicitly.

```
;; lookup : Symbol Env -> Value
(define (lookup [name : Symbol] [env : Env]) : Value
  (type-case (Optionof Value) (hash-ref env name)
    [(none)   (error name "unbound variable")]
    [(some v) v]))
```

6. extend (Env Symbol Value -> Env)

Returns a *new* environment containing all previous bindings plus the new one. The original environment is never modified.

```
;; extend : Env Symbol Value -> Env
(define (extend [env  : Env]
                [name : Symbol]
                [val  : Value]) : Env
  (hash-set env name val))
```

7. Parser (S-Exp -> Exp)

Converts surface syntax into an **Exp** AST node. The parser is identical in both the substitution and environment versions — it only produces the AST.

```
(define (parse [s : S-Exp]) : Exp
  (cond
    [(s-exp-number? s)
     (numE (s-exp->number s))]

    [(s-exp-symbol? s)
     (varE (s-exp->symbol s))]

    [(s-exp-list? s)
     (let ([l (s-exp->list s)])
       (cond
         [(symbol=? '+ (s-exp->symbol (first l)))
          (plusE (parse (second l))
                 (parse (third  l)))]

         [(symbol=? 'let1 (s-exp->symbol (first l)))
          (let* ([binding  (s-exp->list (second l))]
                 [var-name (s-exp->symbol (first  binding))]
                 [var-val  (parse         (second binding))]
                 [body     (parse         (third  l))])
            (let1E var-name var-val body))]

         [else (error 'parse "unknown operator")]))]

    [else (error 'parse "not a number, symbol, or list")]))
```

8. Interpreter (Exp Env -> Value)

Pattern-matches on the AST and returns a **Value**. The key difference from substitution: variables are resolved by **lookup** in the environment, and **let1E** calls **extend** rather than **subst**. The **subst** function is gone entirely.

```
(define (interp [expr : Exp] [env : Env]) : Value
  (type-case Exp expr

    [(numE n)
     (numV n)]

    [(plusE l r)
     (numV (+ (numV-n (interp l env))
              (numV-n (interp r env))))]

    [(varE s)
     (lookup s env)]          ; look up — never rewrite the AST

    [(let1E var val-expr body)
     (interp body
             (extend env                      ; step 3: extended env
                     var
                     (interp val-expr env)))]))  ; step 1 & 2
```

9. Runner (S-Exp -> Value)

Composes **parse** and **interp** into a single entry point. Every program starts from the empty environment.

```
(define (run [s : S-Exp]) : Value
  (interp (parse s) mt-env))   ; always start with empty env
```

10. Tests

```
(print-only-errors #true)

; --- num and plus ---
(test (run `1)               (numV 1))
(test (run `{+ 1 2})         (numV 3))
(test (run `{+ {+ 1 2} 3})   (numV 6))

; --- simple let1 ---
(test (run `{let1 {x 1} x})         (numV 1))
(test (run `{let1 {x 5} {+ x x}})   (numV 10))
(test (run `{let1 {x 5} {+ x 1}})   (numV 6))

; --- nested let1, different names ---
(test (run `{let1 {x 1}
               {let1 {y 2}
                  {+ x y}}})
      (numV 3))

; --- shadowing: inner x shadows outer x ---
(test (run `{let1 {x 1}
               {let1 {x 2}
                  x}})
      (numV 2))

; --- shadowing: outer x still visible ---
(test (run `{let1 {x 1}
               {+ x
                  {let1 {x 2} x}}})
      (numV 3))

; --- unbound variable raises error ---
(test/exn (run `x)       "unbound")
(test/exn (run `{+ x 1}) "unbound")
```

Substitution vs. Environment — Key Differences

Variable resolution	`Rewrite AST with subst`	`Look up table with lookup`
varE in interpreter	`(error s "unbound variable")`	`(lookup s env)`
let1E in interpreter	`Calls subst, then interp`	`Calls extend, passes new env`
subst function	`Required`	`Gone entirely`
interp signature	`Exp -> Number`	`Exp Env -> Value`
Return type	`Raw Number`	`Value (numV)`

Environment	`None`	`mt-env, extend, lookup`
Shadowing mechanism	`subst stops at same-name let1`	`hash-set overwrites the key`

Pipeline Traces

Step-by-step execution of three example S-expressions through the full run → parse → interp pipeline.

Trace 1 — `1

Step 1 · run is called

```
(run `1)
=> (interp (parse `1) mt-env)
```

↓

Step 2 · parse receives `1

```
Input: 1   (s-exp-number? s)  =>  YES
(numE (s-exp->number s))
parse output:  (numE 1)
```

↓

Step 3 · interp receives (numE 1) and mt-env

```
(interp (numE 1) mt-env)
type-case matches:  [(numE n) (numV n)]   where n = 1
env is never consulted — a literal needs no lookup.
interp output:  (numV 1)
```

↓

Final Output

```
(numV 1)
```

Trace 2 — `{+ x 1}

Step 1 · run is called

```
(run `{+ x 1})
=> (interp (parse `{+ x 1}) mt-env)
```

↓

Step 2 · parse receives `{+ x 1}

```
Input is a list:  l = [ '+,  x,  1 ]
                       first  second  third
head is '+  =>  plusE branch fires:
  (plusE (parse (second l))   ; parse x  => (varE 'x)
         (parse (third  l)))  ; parse 1  => (numE 1)
parse output:  (plusE (varE 'x) (numE 1))
```

↓

Step 3 · interp receives (plusE (varE 'x) (numE 1)) and mt-env

```
plusE branch fires:
  (numV (+ (numV-n (interp (varE 'x) mt-env))
           (numV-n (interp (numE 1)  mt-env))))
  -- left side first --
  (interp (varE 'x) mt-env)
    => [(varE s) (lookup s env)]   s = 'x
    => (lookup 'x mt-env)
    => (hash-ref {} 'x)  =>  (none)
    => (error 'x "unbound variable")   *** STOP ***
```

↓

Error Raised

x: unbound variable The right side (numE 1) is never reached. Racket evaluates left-to-right and stops at the first error.

Trace 3 — `{let1 {x 1} {+ x 2}}

Step 1 · run is called

```
(run `{let1 {x 1} {+ x 2}})
=> (interp (parse `{let1 {x 1} {+ x 2}}) mt-env)
```

↓

Step 2 · parse receives `{let1 {x 1} {+ x 2}}

```
Input is a list:  l = [ let1,  {x 1},  {+ x 2} ]
                        first  second   third
head is 'let1  =>  let1 branch fires:
  binding  = (s-exp->list (second l))  =>  [ x, 1 ]
  var-name = (s-exp->symbol (first binding))  =>  'x
  var-val  = (parse (second binding))  =>  (numE 1)
  body     = (parse (third l))
           = (parse `{+ x 2})
           = (plusE (varE 'x) (numE 2))
parse output:
  (let1E 'x
         (numE 1)
         (plusE (varE 'x) (numE 2)))
```

↓

Step 3 · interp receives let1E AST and mt-env

```
(interp (let1E 'x (numE 1) (plusE (varE 'x) (numE 2)))
        mt-env)
let1E branch fires:
  (interp body (extend env var (interp val-expr env)))
```

↓

Step 3a · evaluate val-expr in current env

```
(interp (numE 1) mt-env)
  numE branch:  (numV n)  =>  (numV 1)
result:  (numV 1)
```

↓

Step 3b · extend the environment

```
(extend mt-env 'x (numV 1))
  (hash-set {} 'x (numV 1))
  mt-env  =  {}                      (unchanged)
  new-env =  { 'x -> (numV 1) }      (brand new env)
```

↓

Step 3c · evaluate body in extended env

```
(interp (plusE (varE 'x) (numE 2))
        { 'x -> (numV 1) })
plusE branch fires:
  LEFT:  (interp (varE 'x) { 'x -> (numV 1) })
         => (lookup 'x { 'x -> (numV 1) })
         => (hash-ref env 'x)  =>  (some (numV 1))
         => type-case (some v): returns (numV 1)
  RIGHT: (interp (numE 2) { 'x -> (numV 1) })
         => (numV 2)
  ADD:   (numV (+ (numV-n (numV 1)) (numV-n (numV 2))))
         = (numV (+ 1 2))
         = (numV 3)
```

↓

Final Output

```
(numV 3)
```

Summary of Pipeline Traces

`1	(numE 1)	(numV 1)
`{+ x 1}	(plusE (varE 'x) (numE 1))	ERROR: x unbound
`{let1 {x 1} {+ x 2}}	(let1E 'x (numE 1) (plusE (varE 'x) (numE 2)))	(numV 3)